**EF CORE 6**

**ORM** (Object Relational Mapping) : veri erişim tekniğidir.

**EF Core** da bir ORM tooludur.

**LINQ** (Language Integrated Query) : C# 3.0 ile geldi. Kodda query yazma tooludur.

**DbContext :** DB Modeli (= veritabanı)

**DBSet:** Entity Model (=tablo)

**Avantajları:**

1. Tip güvenli çalışır. Yani kod, compile edilirken hata alır, çalışırken değil.
2. Tüm süreçleri, dönüşümleri EF Core yapar.
3. Okunabilirlik artar.
4. Sürdürülebilir: db değiştikçe kodlar değişir ado.nette, fakat efcore'da değişmez.
5. SQL injection gibi zafiyetlere karşı korur.
6. Tüm db işlemlerini kodda yapabilir.

**Dezavantajları:**

1. Tüm süreç soyutlandığı için daha yavaştır.
2. Querylere %100 müdahele edemeyiz.
3. ilişkisel db lerde çalışır.(SQL Server, Oracle, PostreSql, MySql)
4. ilişkisel olmayanlarda çalışmaz. (MongoDb)

Yaklaşımlar

1. **DB First**

DBFirst'de önce DB'ye gidilir ve tablolar oluşturulur, sonra kodda bu modeller oluşturulur manuel. burada 2 yöntem izleyebiliriz.

1. DBFirst Console Uygulamamızda olduğu gibi

* DBContextInitializer.cs oluşturularak gerekli ayarlar yapılır.
* DB'de oluşturulan her tablonun modeli oluşturulur. Product.cs gibi.
* Daha sonra bu tabloların bulunduğu bir DBContext classı oluşturulur : AppDbContext.cs
* burada class DbContext'ten miras alır => public class AppDbContext: DbContext
* bu classın içerisinde kullanacağımız tüm tabloları DBSet attribute'u ile ekleriz. : public DbSet<Product> Products { get; set; }
* AppDbContext classı içinde 2 farklı constructor oluşturduk.
  + 1. si parametre almayan default constructor. appdbcontext her newlendiğinde onconfiguring metodu da tetiklenerek ayarlar ayağa kalacaktır.
  + 2. constructor optionsları dışardan verme.her appdbcontext newlendiğinde bu optionslar parametre olarak constructora geçilecektir.
  + Her ikisi de aynı amaca hizmet eder
* eğer boş ctor kullanılıcaksa dbcontextInitializer.cs içinde UseSqlServer satırındaki comment kaldırılır.
* ikisinde de kod çalışmadan önce yaptığımız ayarların ayağa kalkması gerekir.
* Program.cs içinde en başta Build() çalışır ve ayarlar ayağa kalkar.
* böylece db'yi kodda ayağa kaldırmış oluruz.

2. DBFirstByScallfold Console Uygulamamızda olduğu gibi

* Hazır DB ve tablolar var ise, uzun uzadıya tablo modelleri oluşturmak yerine DBFirst mantığıyla Scaffold kullanılabilir.
* Microsoft.EntityFrameworkCore.Tools package'i indirilir ki Package Manager Console'da komut çalıştırabilelim.
* Package Manager Console'dan şu sorguyu çalıştırıp tüm tabloların modelleri oluşturulur dbcontexti oluşturulur.
  + Scaffold-DbContext "Data Source={{dbsourcename}}" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
* bütün ayarlar yapılmış olur ve bir daha hiç bir yerde (proje ilk ayağa kalkacağı zaman bile) tekrardan ayarları ayağa kaldır dememize gerek kalmaz.
* bundan sonra codefirst ile devam edilmesi önerilir.
* çünkü her seferinde scaffold çalıştırırsak tüm ayarlar(örn: modele [Required] gibi annotation eklediğinde) bunlar gider ve best practice değildir.

1. **Code First**

* Burada DB'de görmek istediğimiz tabloların tasarımını önce kodda yaparız.
* Örnek olarak bir Product classı oluşturulur hangi kolonların olacağını belirtiriz.Burada Id veya ProductId olarak tanımlanan kolon olursa EFCore bunu otomatik olarak PrimaryKey olarak kabul eder ve birer birer auto-increment şekilde artacağını varsayar.
* Daha sonra DB'deki tüm tabloları bir arada göstereceğimiz bir DBContext oluştururuz.
* Örnek olarak AppDbContext classı oluşturulur ve içine tablolar DBSet ile tanımlanır.
  + DbSet<Product> Products : Tablonun içeriği Product.cs'ten gelecek demek ve db de oluşacak tablonun adı da Products demektir bu.
* Bu bir console uygulaması olduğu için appsettings.json ve config leri bizim yapmamız gerekir.
* Bu yüzden Initializer.cs oluşturulur. AppDbContext ile bu ayarlar ayağa kaldırılır.
* Bundan sonrası artık kodda oluşturulan modelin DB'ye yansıtılması.
* 4 tane migration komutu var. Bunlar şu şekilde:

Mıgratıon

**Migration** : bir tooldur bizim kodumuzla db'yi senkronize halde çalışmasını sağlar.

1. **add-migration**
   1. ***PM> add-migration initial*** : Kodda bir Migration klasörü oluşturulur otomatik olarak. 2 tane cs dosyası içerir. biri verdiğimiz isimde (gününtarihi\_initial) bir class, diğeri de snapshot classı. Gününtarihi\_initial.cs dosyasında up ve down metotları var. Up'da createtable komutunu içerir. AppDbContext içinde tanımladığımız her bir DbSet objesinin komutunu hazırlar. hangi kolonun primary key olacağına da EFCore ya Id yazan attribute'u yada ProductId gibi classın adını içeren attribute'u belirler. Down metotunda da up metotundakinin tam tersi komutunu içerir. yani ilk kez migration yaptığımızda up'ta createtable içeriyorken down'da droptable içerir. updateler yapılınca up'da yeni tipine dönüştüren sorgu varken, downda eski tipine döndüren sorgu olur. Snapshot classında ise db'de hangi kolonların hangi tipte create edileceğini içerir. modelimizde yaptığımız her değişiklikte (yeni property oluşturulunca, kolon değişince) bu metot çalışır.
2. **update-database** 
   1. ***PM> update-database*** : add-migration ile oluşmuş migrationu db'de oluştur db'ye yansıt demektir. gununtarihi\_name classındaki up metotunu çağırır.
3. **remove-migration**
   1. ***PM> remove-migration*** : db ye yansıtılmamış son yapılan migrationu silmek için kullanırız. bu durumda gununtarihi\_name classındaki down metotunu çağırır yada db ye yansıtılmış bir migrationda önce istenen migration versiyonuna update-database metotuyla eski versiyona dönülür ve remove-migration ile son migration silinir.
4. **script-migration**
   1. ***PM> script-migration*** : oluşturulmuş migration dosyalarının sql deki script dosyasını oluşturur.

DBContext

veriye tüm erişimi soyutlayan yapısından dolayı => repository pattern

içinde default transaction yapısı olduğundan dolayı => unit of work pattern

neden generic repository yada uow kullanırız?

* çok fazla entity olduğunda temel crud operasyonlarını tekrar yazmamak
* kullanmış olduğumuz mimarinin best practicelerini uygulamak

**Metotları:**

* *Add/AddSync*
* *Update* = normalde track edilen bir nesnenin bir propertysine yeni bir değer atarsak efcore update olarak anlardı, fakat biz hiç track etmediğimiz manuel olarak id yi verdiğimiz bir nesneyi update etmek istediğimizde bu çalışır. EFCORE tarafından track edilen bir entity'de güncelleme yapıldığında tekrar update() metotunu çağırmamıza gerek yok.
* *Remove*
* *Find/FindAsync*
* *SaveChanges/SaveChangesAsync*

**state types:** efcore da yapılan işlemler önce db de tracker olarak bazı statülerle yazılır.

var state = \_context.Entry(p).State;

* *added*
* *modified*
* *deleted*
* *unchanged* : efcore'un ilkkez çektiği bir kayda verdiği state veya savechanged dan sonra gelen ilk state. yani db ile dbcontext aynı demek
* *detached :* efcore un track edilmediği kayda veya artık db den silinmiş kayıtlara verdiği state

**properties:**

* *ChangeTracker* : memoryde track edilen datalara erişmemizi sağlar.

\_context.ChangeTracker.Entries().ToList();

* *ContextId (Logging)* : her bir dbcontext e verdiği id dir.
* *Database (connection, command, transaction, raw sql)* : veritabanı ile ilgili genel işlemlerde bu propertyi kullanırız.

DBSET<>

**Metotları:**

* *Add/AddSync:*
* *Update:*
* *Remove:*
* *AsNoTracking* : sqlden gelen datanın track edilmemesini sağlar, memory de tutulmaz, propertyleri değişmicekse kullanılır.
* *Find /FindAsync* : priary key ile aramada kullanılır.
* *FirstOrDefault* : arana kriterlerde bir yada birden fazla data varsa ilkini döner yoksa null döner.
* *SingleOrDefault* : aranan kriterde birden fazla kayıt varsa hata fırlatır. yada hiç kayıt yoksa null döner. aranan kriterde tek bir data varsa onu döner.
* *First* : şartı sağlayan bir tane kayıt varsa döner, yoksa hata fırlatır.
* *Single* : şartı sağlayan bir tane kayıt varsa döner, yoksa yada birden fazla varsa hata fırlatır.
* *Where*

|  |  |  |
| --- | --- | --- |
| DataAnnotations | FluentApi | Convensions |
| [Table] | ToTable | DbSet<Product> Products |
| [Column] | HasColumnName | PropertyName |
| [Key] | HasKey | Id veya <Entity>Id |
| [Required] | IsRequired | ValueType |
| [MaxLength(150)] | HasMaxLength | x |
| [StringLength(150,MinimumLength)] | HasMaxLength | x |
| X | IsFixedLength | X |

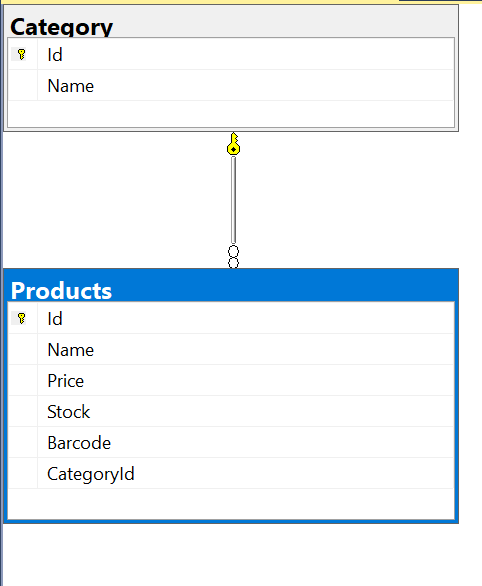
Relatıonshıps

Bazı kavramlar:

* Parent => Principal entity
* Child => dependent entity
* Primary Key => Principial key
* Foreign Key
* Navigation property
* Shadow property

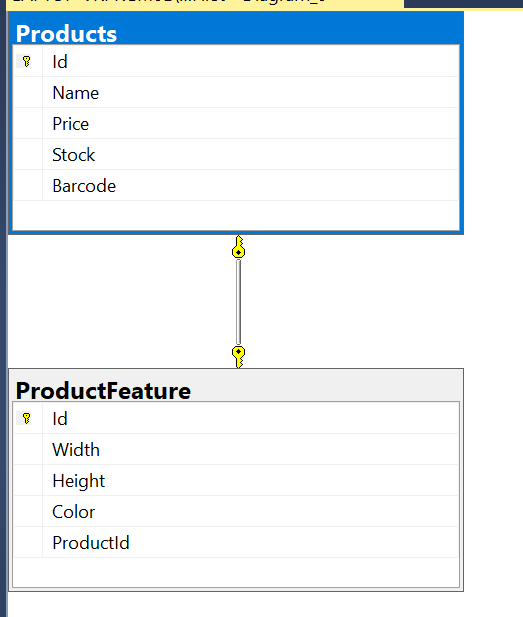
1. **One-to-Many**

* Örnek => Product – Category
* Parent : Category
* Child : Product
* Yöntem1 : Convension
  + productın 1 categorysi var.
    - public int CategoryId { get; set; } //tabloadıUniqueKey şeklinde tanımladığımız için efcore bunun foreign key olduğunu anlar ve hiç fluentapi kod yazmayız.
    - public Category Category { get; set; } //navigation property
  + Her categorynin n productu var.
    - public List<Product> Products { get; set; } //navigation products
  + add-migration ve update-database ile ilişki db ye de yansır.
* Yöntem2: fluentapi
  + productın 1 categorysi var.
    - public int Category\_Id { get; set; }
    - public Category Category { get; set; } //navigation property
  + Her categorynin n productu var.
    - public List<Product> Products { get; set; } //navigation products
  + appDbContext içine girer OnModelCreating de ilişkiyi tanımlarız.
    - modelBuilder.Entity<Category>().HasMany(x => x.Products).WithOne(x => x.Category).HasForeignKey(x => x.Category\_Id);
* Yöntem3: attributebased
  + productın 1 categorysi var.
    - public int Category\_Id { get; set; } [ForeignKey("Category\_Id")]
    - public Category Category { get; set; } //navigation propertyHer
  + categorynin n productu var.
    - public List<Product> Products { get; set; } //navigation products



1. **One-to-one**

* Örnek : Product – ProductFeature
* Parent: Product
* Child : ProductFeature
* Yöntem1 : convension
  + 1 product’ın 1 productfeature’ı olur.
    - public ProductFeature ProductFeature { get; set; }
  + 1 product feature’un 1 product’ı olur.
    - public int ProductId { get; set; } // foreign key : parent unique key
    - public Product Product { get; set; } //navigation property
* Yöntem2: fluentapi
  + 1 product’ın 1 productfeature’ı olur.
    - public ProductFeature ProductFeature { get; set; }
  + 1 product feature’un 1 product’ı olur.
    - public int ProductRef\_Id { get; set; } // foreign key : parent unique key
    - public Product Product { get; set; } //navigation property
  + appDbContext içine girer OnModelCreating de ilişkiyi tanımlarız.
    - modelBuilder.Entity<Product>().HasOne(x=> x.ProductFeature).WithOne(x=>x.Product).HasForeignKey<ProductFeature>(x=> x.ProductRef\_Id);
* Yöntem3: attribute based
  + 1 product’ın 1 productfeature’ı olur.
    - public ProductFeature ProductFeature { get; set; }
  + 1 product feature’un 1 product’ı olur.
    - public int ProductRef\_Id { get; set; } // foreign key : parent unique key
    - [ForeignKey("ProductRef\_Id")]
    - public Product Product { get; set; } //navigation property



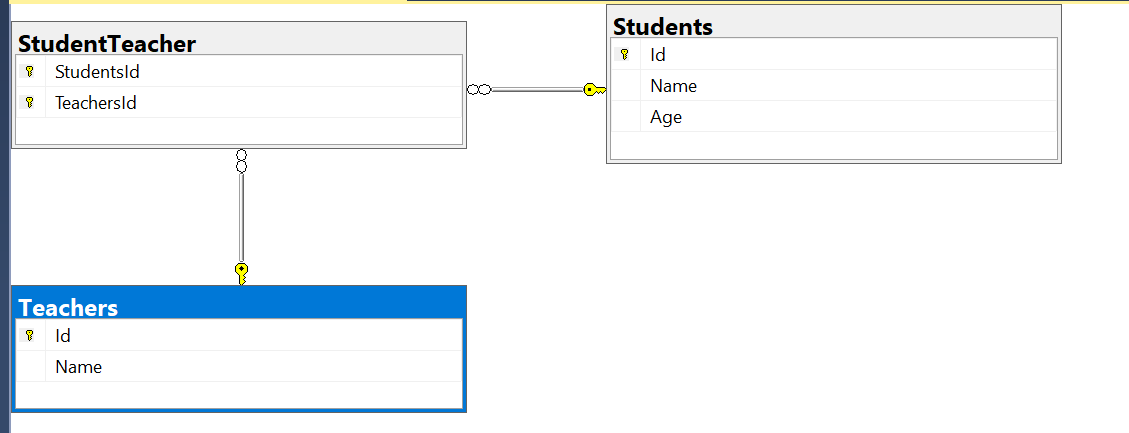
1. **Many-to-Many**

* Örnek : Student-Teacher
* Yöntem1 : convension
  + 1 student’ın n tane teacher’ı olabilir.
    - public List<Teacher> Teachers { get; set; }
  + 1 teacher’ın n tane student’
    - public List<Student> Students { get; set; }
* Yöntem2: fluentapi
  + 1 student’ın n tane teacher’ı olabilir.
    - public List<Teacher> Teachers { get; set; }
  + 1 teacher’ın n tane student’
    - public List<Student> Students { get; set; }
  + appDbContext içine girer OnModelCreating de ilişkiyi tanımlarız.
    - modelBuilder.Entity<Student>()

.HasMany(x=>x.Teachers).WithMany(x=>x.Students)

.UsingEntity<Dictionary<string, object>>(

"StudentTeacherManytoMany", // customize relation table name x=>x.HasOne<Teacher>().WithMany().HasForeignKey("Teacher\_Id").HasConstraintName("FK\_TeacherId"), // customize foreign key x=>x.HasOne<Student>().WithMany().HasForeignKey("Student\_Id").HasConstraintName("FK\_StudentId")); // customize foreign key



Veri ekleme örneği:

using (var context = new AppDbContext())

{

//ONE-TO\_MANY DATA ADDING EXAMPLE

var category = new Category() { Name = "Kalem" };

var product1 = new Product()

{

Name ="dolma kalem",

Price=3,

Stock=200,

Barcode="123",

Category=category

};

context.Products.Add(product1);

//ONE-TO-ONE DATA ADDING EXAMPLE

//Product => Parent

//Child => ParentFeature

var product2 = new Product()

{

Name = "kurşun kalem",

Price = 3,

Stock = 200,

Barcode = "123",

Category = category,

ProductFeature = new ProductFeature()

{

Color="Red",

Height=4,

Width=4

}

};

context.Products.Add(product2);

//MANY-TO-MANY DATA ADDING EXAMPLE

var student = new Student() { Name="Ahmet", Age=19 };

student.Teachers.Add(new() { Name = "fatma öğretmen" });

student.Teachers.Add(new() { Name = "ali öğretmen" });

context.Add(student);

context.SaveChanges();

}

Delete Behavıors

* **Cascade** : default gelen davranıştır. Category silinirse, bağlı olan productlardan tablodan silinir.
* **Restrict**: eğer categorye bağlı productlar varsa categorynin silinmesini engeller.
* **SetNull** : child tablosunda foreignkey nullable ise yani product tablosundaki categoryid nullable ise, category tablosundan silinen bir kayıt olursa onun bağlı olduğu productların categoryid sini null olarak set eder.
* **NoAction** : hiçbir şeye karışmaz.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//fluent api kodları buraya yazılıyor.

modelBuilder.Entity<Category>()

.HasMany(x=> x.Products)

.WithOne(x=>x.Category)

.HasForeignKey(x=>x.CategoryId)

.OnDelete(DeleteBehavior.Restrict);

base.OnModelCreating(modelBuilder);

}

Database Generate Attrıbute

* **Computed** : EF Core add ve update işlemlerinde bu alanı sorgulara dahil etmez.
* **Identity**: EF Core sadece update işlemlerinde bu alanı dahil etmez.
  + [DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public DateTime? CreatedTime { get; set; } = DateTime.Now;

* **None** : Veritabanı tarafından otomatik değer üretmeyi kapatır.

Fluentapi:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//fluent api kodları buraya yazılıyor.

modelBuilder.Entity<Product>().Property(x => x.PriceKDV)

.HasComputedColumnSql("[Price]\*[KDV]");

modelBuilder.Entity<Product>().Property(x => x.PriceKDV)

.ValueGeneratedOnAddOrUpdate();//computed

.ValueGeneratedOnAdd(); //identity

.ValueGeneratedNever(); //None

base.OnModelCreating(modelBuilder);

}

Related Data Load

1. **EAGER LOADING**

Bir tabloyu çekerken ona bağlı nesneleri de getirir.

//Include ile Category'nin product ile ilişkilerini çekeriz.

var categoryWithProducts = context.Categories.Include(x=>x.Products).First();

//ThenInclude ile Product’ın productfeature ile ilişkilerini çekeriz.

var categoryWithProducts = context.Categories.Include(x=>x.Products).ThenInclude(x=>x.ProductFeature).First();

1. **EXPLICIT LOADING**

//şimdi sadece category bilgisini çekeriz. ona bağlı ilişkileri çekmeyiz.

var \_category = context.Categories.First();

//sonra iş kural gereği yada performans sorunu yaratmaması için ilerde uygun bir case'de ilişkilerini çekmek için şöyle yaparız.

//category'nin n tane product ilişkisi olabilir diye Collection kullanırız.

context.Entry(\_category).Collection(x=>x.Products).Load();

var \_product = context.Products.First();

//productın 1 tane productfeature'u olduğu için Reference metotu kullanılır.

context.Entry(\_product).Reference(x=>x.ProductFeature).Load();

1. **LAZY LOADING**

Explicit loading gibidir. Başta nesneyi çekerken ilişkilerini getirmeyiz daha sonra ihtiyaç durumunda ilişkilerini çekeriz. Ama explicitte ayrıca kod yazıyoruz (Collection, reference vs.) lazy loadingde yazmıyoruz. Bunu kullanmak için :

1. Microsoft.EntityFrameworkCore.Proxies kütüphanesini indiririz.
2. Lazyloading özelliğini açarız.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

Initializer.Build();

optionsBuilder.UseLazyLoadingProxies().UseSqlServer(Initializer.Configuration.GetConnectionString("SqlCon"));

}

1. Tüm Navigation propertyleri de virtual olarak işaretleriz.

public virtual Category Category { get; set; }

public virtual ProductFeature ProductFeature { get; set; }

koddan çağırım şu şekildedir:

var \_category2 = context.Categories.First();

var \_products2 = \_category2.Products;

dezavantajı her yeni alt bilgiyi çekmek istediğimizde her seferinde db ye istek atar. Performans sorunu yaratır. N+1 problemi olarak anılır. Probleme örnek:

foreach(var item in \_products2)

{

var productFeature = item.ProductFeature;

}

INHERITENCE

1. **Tph (table-per-hierarchy)**

burada 2 yöntem vardır.

1. Base class, dbcontext classında hiç belirtilmez ve child classlar base classtan miras alarak ayrı ayrı tabloları db de oluşur, ama base classın oluşmaz. Örnek olarak:

Person classı : içinde name, surname, age içerir. Base classtır.

Employee classı: person classından miras alır ve ek olarak salary propertysi bulunur.

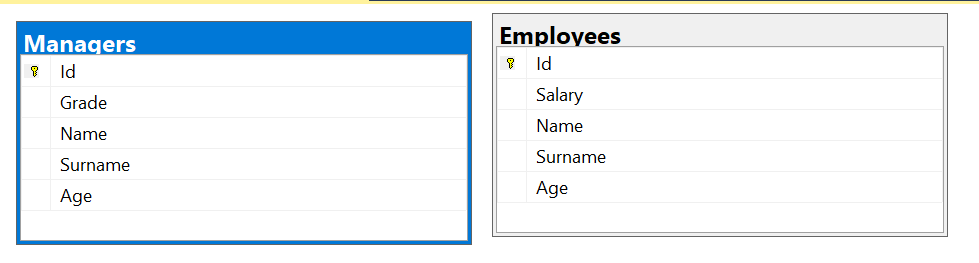
Manager classı : person classından miras alır ve ek olarak grade propetysi bulunur.

Db contextten çağırım:

public DbSet<Manager> Managers { get; set; }

public DbSet<Employee> Employees { get; set; }

Db de görünüm: child classlar oluşur. Baseden aldığı tüm attributeları içerirler:



Data ekleme şekli:

context.Managers.Add(new Manager() { Name="m1",Surname="m2", Age=25, Grade=1});

context.Employees.Add(new Employee() { Name = "e1", Surname = "e2", Age = 20, Salary = 1000 });

Data okuma şekli:

var managers = context.Managers.ToList();

var employees = context.Employees.ToList();

1. Base class da db context classında belirtilir. Bu durumda db de tek bir tablo olıuşur. Base classın adında bir tablo oluşur ve onu miras alan tablolardaki diğer propertyler de burada oluşur. Ek olarak da ef core discriminator diye bir kolon oluşturur ve hangi entityden geldiğini tutar. Örnek olarak:

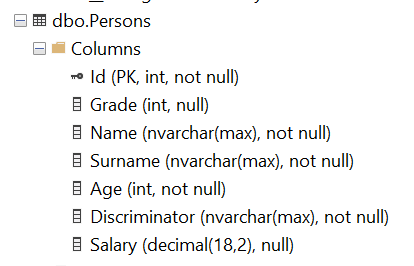
Db contextten çağırım:

public DbSet<Person> Persons { get; set; }

public DbSet<Manager> Managers { get; set; }

public DbSet<Employee> Employees { get; set; }

db de görünümü: tek bir tablo oluşur o da base tablo. Diğerlerine ait propertyler de burada.



Data ekleme şekli:

context.Persons.Add(new Manager() { Name = "m1", Surname = "m2", Age = 25, Grade = 1 });

context.Employees.Add(new Employee() { Name = "e1", Surname = "e2", Age = 20, Salary = 1000 });

1. **Tpt (table-per-type)**

Base class ve tüm childlar da db de oluşsun istiyorsak burada OnModelCreating metotunda kod yazmamız gerek.

//TPT

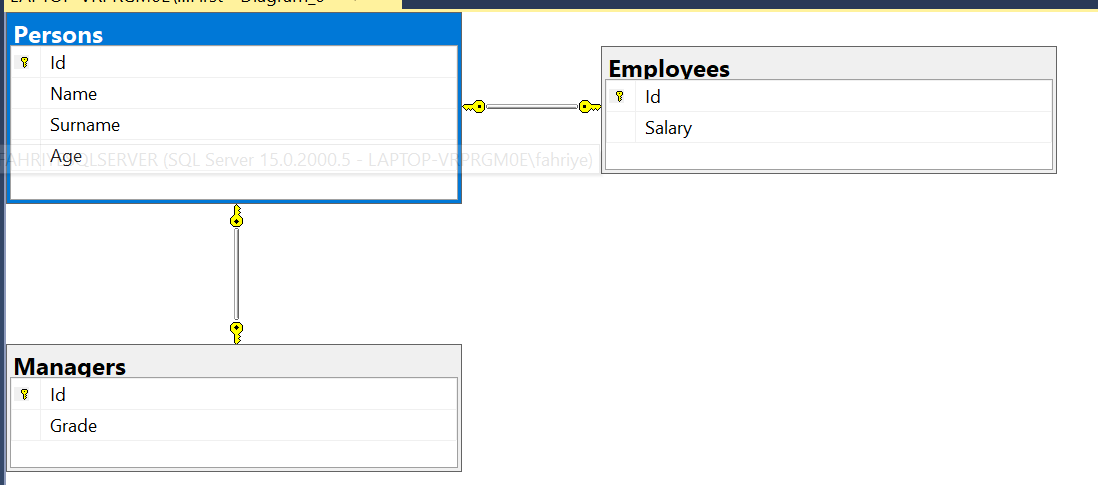
//modelBuilder.Entity<{classname}>().ToTable("{tablename}");

modelBuilder.Entity<Person>().ToTable("Persons");

modelBuilder.Entity<Manager>().ToTable("Managers");

modelBuilder.Entity<Employee>().ToTable("Employees");

db görünümü:

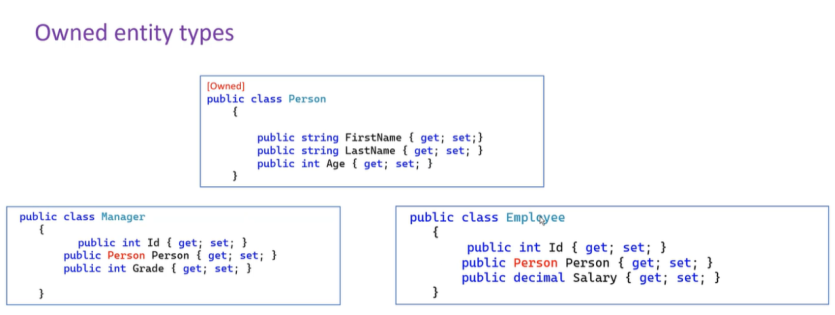


Model

1. **Owned Entıty Types [Owned]**

Bir inheritance kullanımı olmadan ortak attributelar kullanılmak istendiğinde Person gibi ortak bir class oluşturup bunu owned olarak işaretlediğimizde Person classı tablo olarak db de oluşmaz ama bunu kullanan tüm modellerde kolon olarak oluşur.

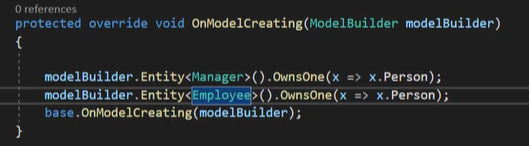
* Data annotation ile



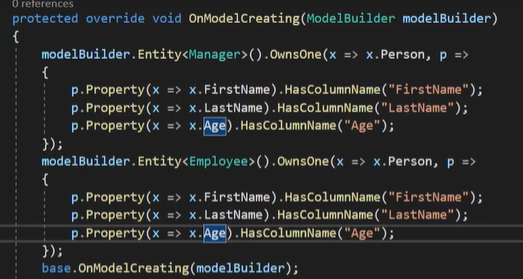
Db görüntüsü:



* Fluent api ile



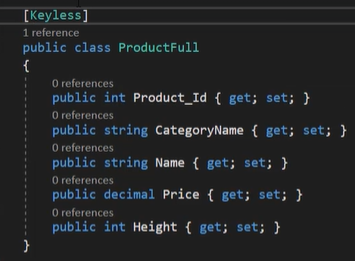
Customize etmek istersek kolonları:



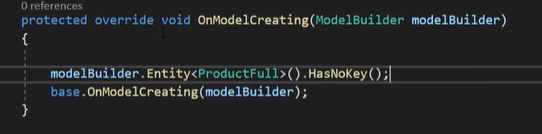
***Keyless Entıty Types [Keyless]***

Key tanımlı olmayan entityleri efcore track etmez, insert/update/delete yapılmaz ya da Sql den dönen rawsql sorgularını maplerken kullanırız.

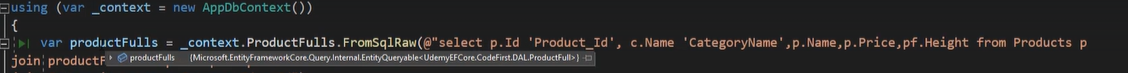
* Data annotation ile



* Fluentapi ile



Bunlar sayesinde primary keysiz bir entity ayağa kaldırabiliriz ve raw sql sorgusunu bu modele set edebiliriz.

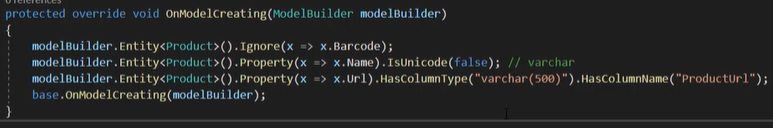


Migration ile bu ProductFull entitysi db de bir tablo olarak oluşur fakat bunu insert/update/delete edemeyiz.

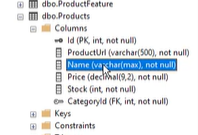
Entıty propertıes

Data annotation using ⬄ fluent api using

* [NotMapped] data ⬄Ignore() : kolonun db de oluşmasını engeller.
* [Column(“Name”)]: [Column(TypeName=”nvarchar(200)”)] ⬄hascolumnName(“name”).hasColumnType(“varchar(200)”) : classtaki attribute’u db de customize edip istenilen name ve columntype ile create eder.
* [Unicode(false)] : varchar ⬄ Isunicode(false);



Db görünütüsü:

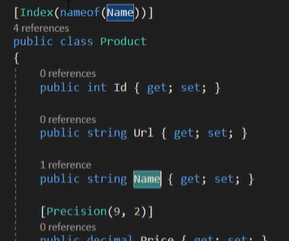


Indexes:

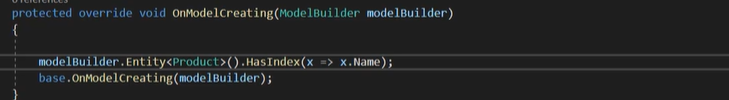
* Efcore da primarykeylere clusterindex atanır
* Foreignkey için otomatik index atanır.
* Bunlar dışında kolonlara sorgu atıyorsak kendimiz index oluştururuz.
* Bunu db de yapacağımız gibi efcore da yapabiliriz.

1. ***Normal ındex***

* Data annotation ile :

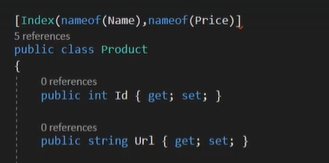


* Fluent api ile:

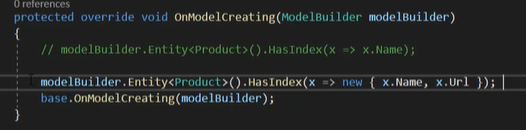


1. ***Composed ındex***

* Data annotation ile:



* Fluent api ile:



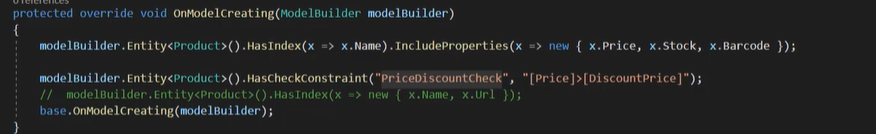
1. ***Included column***

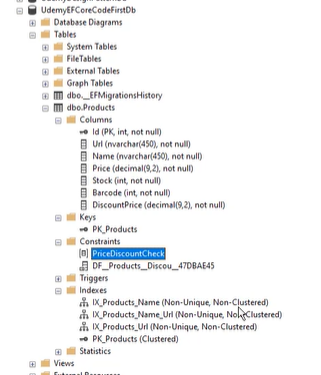
* Index ile çekilecek dataya ek olarak sürekli istenen kolonlarında index tablosundan gelmesini sağlamak için bu metot kullanılır. Eğer bu metot kullanılmazsa her sorgu atışımızda önce index tablosuna gider kalem1 adında bir index bulup onun tablodaki yerini öğrenir sonra product tablosuna gidip istenen alanları çekmeye çalışır. Ama bunun da dezavantajı var tabi. Index’le beraber bu istenen kolonları da index tablolarında tutacağı için index için ayrılan storage artar.



1. ***Check constraınts***

* Burada db nin tutarlılığını sağlamak ve olası kodda gerekli kontrollerin yapılmadığı durumlarda db de constraint ekleyerek veri bütünlüğü sağlanabilir.
* Fluent api ile:





QUERY

Client vs Server Evaluation

Server Evaluation = Veritabanına gönderilecek olan sql cümleciğidir. Lokal fonksiyon içermez.

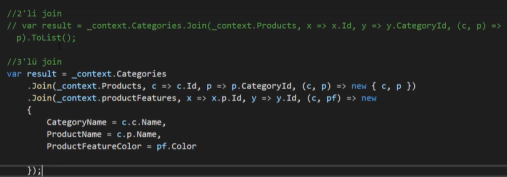
Client Evaluation = Memoryde gelen data üzerinde sorgulama yaptığımız kısımdır. Lokal fonksiyon içerebilir.

Joins:

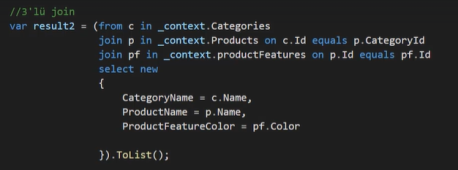
İki entity arasında navigation propertyler varsa lazyloading, expilicit loading gibi yöntemler kullanırız. Navigation property yoksa join kurarız.

1. ***Inner join :***

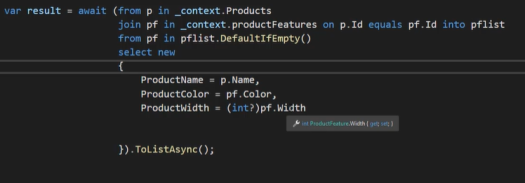
1.yol:



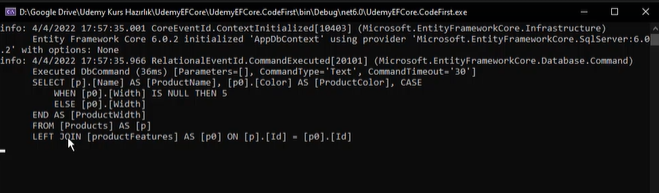
2.yol:



1. ***Left join:***

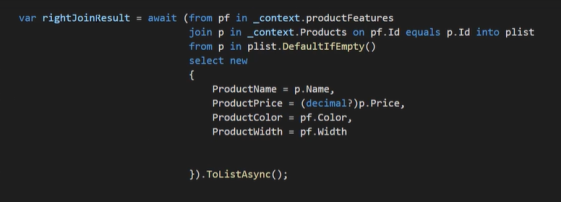


Arka planda efcore’un oluşturduğu sorgu:



1. ***Right join:***

Right left join metotları yok sadece yukardaki sorguda tabloların yerlerini değiştiririz.



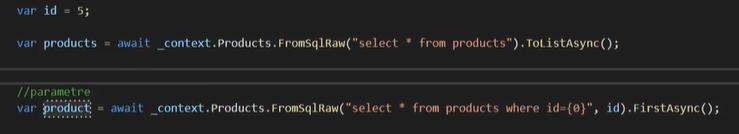
1. ***Full outer join:***

Bunun içinde efcore da bir metot yok bunun yerine left join sorgumuz + right join sorgumuz şeklinde yapıyoruz.



Raw sql

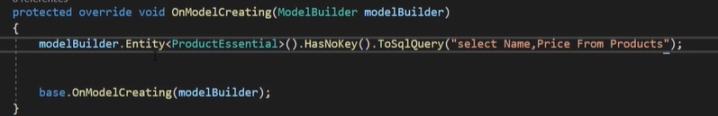
1. ***FromSqlRaw()***



1. ***FromSqlInterpolated()***



1. ***ToSqlQuery()***

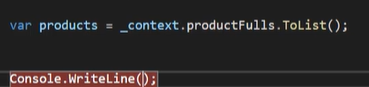
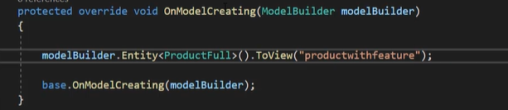


Sonrasında productessential’ı şöyle hiçbir sql sorgucuğu yazmadan çekebilecez:



1. ***ToView()***

Dbde bulunan bir view’ı çekip bir modele set ederken toView() metodunu kullanırız.



Pagınatıon

***Take() & Skip()***

List<Person> getProducts(int page, int pageSize){

using (var context = new AppDbContext())

{

//page:1, pageSize:3 => ilk 3 data => Skip:0, Take:3 => Skip=(page-1)\*pageSize => (1-1)\*3 = 0\*3

//page:2, pageSize:3 => ikinci 3 data => Skip:1, Take:3 => Skip=(page-1)\*pageSize => (2-1)\*3 = 1\*3

//page:3, pageSize:3 => üçüncü 3 data => Skip:2, Take:3 => Skip=(page-1)\*pageSize => (3-1)\*3 = 2\*3

return context.Persons.OrderByDescending(x => x.Id).Skip((page-1)\*pageSize).Take(pageSize).ToList();

}

}

***TagWith()*** metodu ile sorguya comment eklemiş olunur.

Stored Procedure (sp)

* Yazdığımız sp, entity modellerimizden birini dönecekse, her hangi bir input almıyorsa, custom bir model dönmüyorsa şu şekilde çağırırız.

var persons = context.Persons.FromSqlRaw("exec sp\_get\_persons").ToList();

* Parametre almayan ama Custom kolonlar dönen bir sp varsa, bu custom entityi dbcontext class içinde bir entity olarak tanımlarız, bunu onModelCreating metodunda HasNoKey() ile işaretleriz ki burada insert/update/delete yapılmayacak ve notracking bir model olduğunu belirtiriz. Migration classlarında bu modelin create sorgusu silinirki db de oluşmasın. Çünkü bu customize entity, aslında db de karşılığı olan bir tablodan ziyade, bize gelen datayla maplenen bir model. Yine veri yukardaki gibi çekilir. Örnek olarak aşağıdaki sorgudaki gibi çekilir: PersonDetails adında bir class oluşturduk, DbContext classımızda Dbset olarak belirtip, OnCreatingModelde HasNoKey() ile işaretledik, migrationda da bu modelin create’ini sildik.

var personDetails = context.PersonDetails.FromSqlRaw("exec sp\_get\_persondetails").ToList();

* SP den sonra where koşulu eklenemez. Functionlara eklenebilir.
* Parametre alan bi sp çağırırsak, yine customize class oluşturulur. Parametreyi FromSqlInterpolated metotuyla gönderiririz. Bu arada FromSqlRaw ile de çağırabiliriz ama parametre geçiş tipi olarak FromSqlInterpolated daha bestpractice.

Var personId = 1;

var personDetails = context.PersonDetails.FromSqlInterpolated($"exec sp\_get\_persondetailswithid {personId}").ToList();

* Insert/Update/Delete yapan sp’lerde bir modelle maplememize gerek yok çünkü en fazla id döneceğiz yada hiçbirşey dönmeyeceğiz.

//insert sp

var person = new Person

{

Name = "ali",

Surname = "veli",

Age = 25

};

var newPersonIdParam = new SqlParameter(@"newId", System.Data.SqlDbType.Int);

newPersonIdParam.Direction = System.Data.ParameterDirection.Output;

context.Database.ExecuteSqlInterpolated($"exec sp\_insert\_person {person.Name},{person.Surname}, {person.Age}, {newPersonIdParam} out");

var newPersonId = newPersonIdParam.Value;

Functıon

* Hiçbir input almayan ve customize edilmiş bir tablo dönen fonksiyonda, customize alanları içeren model create edilir, dbcontext içinde belirtilir. OnModelCreating e gelince burada hasNoKEY İle işaretlememize gerek yok o sp ler için gerekirdi burada ToFunction() metotunu kullanırız. Migrationda yine bu entity db de oluşmasın diye sileriz.

modelBuilder.Entity<PersonDetail>().ToFunction("fc\_getpersondetail");

koda geçince bu functionu aşağıdaki gibi çağırabiliriz.

context.PersonDetail.ToList();

* Parametre alan ve tablo dönen function için yine custom model oluşturulur, dbcontexte eklenir ve sp tetiklenmesi gibi çağırırız.

Var personId = 1;

var personDetails = context.PersonDetails.FromSqlInterpolated($"exec fc\_persondetailswithid {personId}").ToList();

yada dbContext içinde PersonDetails’i dbset olarak belirtmeden onun yerine Dbcontext class içine şöyle yazsak:

public IQueryable<PersonDetail> GetPersonDetails(int personId)

{

return FromExpression(()=> GetPersonDetails(personId));

}

Ve onModelCreating metotuna şunu yazarsak:

modelBuilder.HasDbFunction(typeof(AppDbContext).GetMethod(nameof(GetPersonDetails), new[] { typeof(int) })!)HasName("fc\_personDetails");

koddan direkt to list ile çağırırız.

var person2 = context.GetPersonDetails(1).ToList();

* Scalar değer dönen functionlarda: yine table dönen functionun dbcontextteki tanımları gibi:

public int GetPersonCount(int id)

{

throw new NotSupportedException("bu metodu direkt kullanamazsınız");

}

//return scalar value

modelBuilder.HasDbFunction(typeof(AppDbContext).GetMethod(nameof(GetPersonCount), new[] { typeof(int) })!).HasName("fc\_personDetailsWithId");

var count = context.GetPersonCount(1); //hata fırlatır.

context.Persons.Select(x => new

{

Name = x.Name,

PersonCount = context.GetPersonCount(x.Id)

}); // doğru yolu

Eğer yukardaki hata fırlatan satır gibi tek satırda kullanmak istiyorsak ve hata almak istemiyorsak dönen scalar value tipinde bir attribute’u bir entity de oluşturup bunu dbset olarak dbcontexte belirtilir, hasnokey ile işaretlenir.

public class PersonCount

{

public int Count { get; set; }

}

public DbSet<PersonCount> PersonCounts { get; set; }

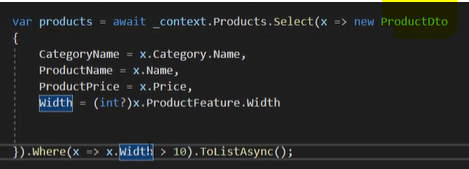
modelBuilder.Entity<PersonCount>().HasNoKey();

int personID = 1;

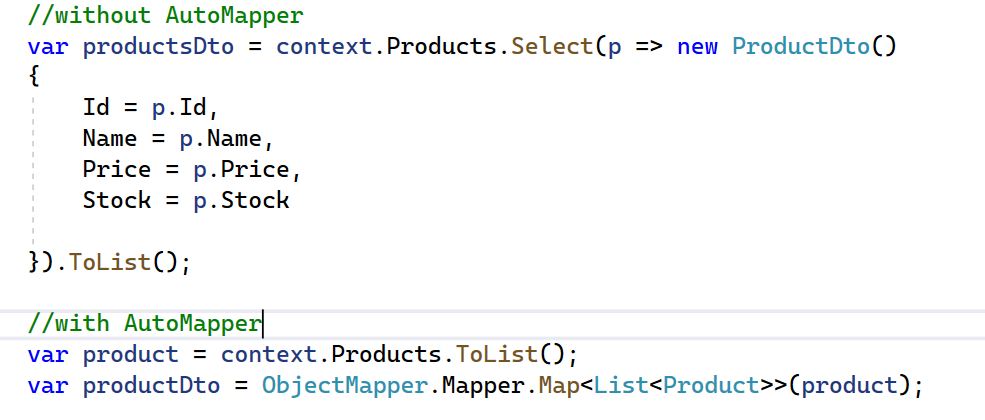
var personCount = context.PersonCounts.FromSqlInterpolated($"select dbo.fc\_get\_personcount({personID}) as Count").First().Count;

Projectıons

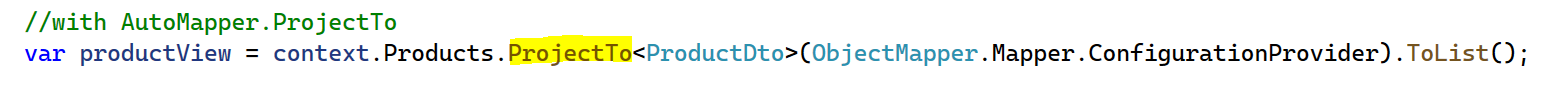
1. Entity
2. DTO/View Model : ProductDto diye customize ve dbcontextt bulunmayacağı bir model oluşturulur.



Böyle uzun uzadıya bir modele maplemek yerine AutoMapper kütüphanesi kullanılabilir. Bu avantajı. Dezavantajı maplenmeyecek diğer tüm alanlar db den çekilir.



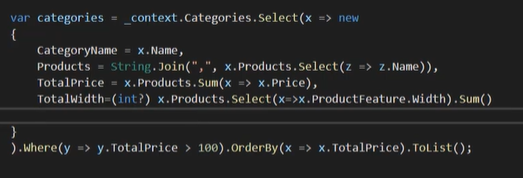
Bu dezavantajı çözmek için ProjectTo() metotu kullanılır. Böylece istenen kolonlar, lazyloadingle istendiği zaman sql’e dönüştürülüp maplenir.



1. Anonymous Types : select ile anonim model oluşturulur.



* Select cümleciği yazdığımızda Include(), ThenInclude() metotlarını kullanmaya gerek yok, select ile çekilen her bir navigation propertyi efcore otomatik çeker.



Kullanılan Kütüphaneler

Microsoft.EntityFrameworkCore => bununla EF Core'u kullanabilicez.

Microsoft.EntityFrameworkCore.SqlServer => SqlServer db sine erişerek EF Core'a bağlayabilecez.

Microsoft.EntityFrameworkCore.Tools => Scaffold, Migration gibi komutları Package manager Console'da yazmamızı sağlayacak.

Microsoft.Extensions.Configuration => kendimiz manuel bir appsetting.json da konfigleri tuttuğumuz için bu json dosyasındaki keyvalue configleri okuruz bu kütüphaneyle

Microsoft.Extensions.Configuration.FileExtensions

Microsoft.Extensions.Configuration.Json

AutoMapper

eklenen appsettings.json dosyalarının her seferinde derlenen exe dosyalarına dahil edilmesi için appsettings.json dosyasını sağ tıklayıp

properties -> copy to output direction = Copy Always yapmamız gerekir. bin-debug altında da oluşturur.